CONSTRUCTORS AND DESTRUCTORS

MODULE 3

Content

- Constructors
  - Default Constructor
  - Copy Constructor
- Destructor

Initializing Objects: Constructors

- The class designer can guarantee initialization of every object by providing a special member function called the constructor.
- The constructor is invoked automatically each time an object of that class is created (instantiated).
- These functions are used to (for example) assign initial values to the data members, open files, establish connection to a remote computer etc.
- The constructor can take parameters as needed, but it cannot have a return value (even not void).

Initializing Objects: Constructors

- The constructor has the same name as the class itself.
- Constructors are generally public members of a class.
- There are different types of constructors.
- For example, a constructor that defaults all its arguments or requires no arguments, i.e. a constructor that can be invoked with no arguments is called default constructor.
- In this section we will discuss different kinds of constructors.

Default Constructors

> A constructor that defaults all its arguments or requires no arguments, i.e. a constructor that can be invoked with no arguments.

```
class Point{
  int x,y;
 public:
  // Declaration of the default constructor
  Point();
  // A function to move points
  bool move(int, int);
  // to print coordinates on the screen
  void print();
};
```

Default Constructors

```
// Default Constructor
Point::Point() {
 cout << "Constructor is called..." << endl;
 x = 0;     // Assigns zero to coordinates
 y = 0;
}

int main() {
   // Default construct is called 2 times
   Point p1(), p2;
   // Default construct is called once
   Point *pp = new Point;
   Point *qq = new Point();
```

Constructors with Parameters

> Like other member functions, constructors may also have parameters.

> Users of the class (client programmer) must supply constructors with necessary arguments.

```
class Point{
    int x,y; coordinates
 public:
    Point(int, int);
    bool move(int, int);
    void print();
};
```

> This declaration shows that the users of the **Point** class have to give two integer arguments while defining objects of that class.

Constructors with Parameters

```
Point::Point(int x_first, int y_first) {
  cout << "Constructor is called..." <<endl;
  x = ( x_first < 0 ) ? 0 : x_first;
  x = ( y_first < 0 ) ? 0 : y_first;
}
int main() {
    Point p1(20, 100), p2(-10, 45);
    Point *pp = new Point(10, 50);
    Point p3;  // ERROR!
    :
}
```

Constructor Parameters with Default Values

> Constructors parameters may have default values

```
struct Point{
  Point(int x_first = 0,int y_first = 0);
  :
};
Point::Point(int x_first,  int y_first) {
   x =  ( x_first < 0 ) ? 0 : x_first;
   y =  ( y_first < 0 ) ? 0 : y_first;
}
```

Constructor Parameters with Default Values

> Now, client of the class can create objects

```
Point p1(15,75);     // x=15, y=75
Point p2(100);       // x=100, y=0
```

> This function can be also used as a default constructor

```
Point p3;   // x=0, y=0
```

Multiple Constructors

> The rules of function overloading is also valid for constructors.
  - A class may have more than one constructor with different type of input parameters:

```
Point::Point() { // Default constructor
    ....// Body is not important
}
// A constructor with parameters
Point::Point(int x_first, int y_first) {
    ... // Body is not important
}
```

Multiple Constructors

> Now, the client programmer can define objects in different ways:

```
// Default constructor is called
Point p1;
// Constructor with parameters is called
Point p2(30, 10);
```

> Following statement causes compiler error, because class does not include a constructor with only 1 parameter.

```
// ERROR!
// There isn't a constructor with 1 parameter
Point p3(10);
```

Initializing Arrays of Objects

> When an array of objects is created, the default constructor of the class is invoked for each element (object) of the array one time.

```
// Default constructor is called 10 times
Point array[10];
```

> To invoke a constructor with arguments, a list of initial values should be used.

> To invoke a constructor with more than one arguments, its name must be given in the list of initial values, to make the program more readable.

Initializing Arrays of Objects

```
// Constructor
Point(int x_first,  int y_first = 0)  {
    ....
}
// Can be called with one or two args
// Array of Points
Point array[]={{10},Point(20),Point(30,40)};
```

> Three objects of type Point has been created and the constructor has been invoked three times with different arguments.

Objects:	**Arguments**:
`array[0]`	`x_first = 10 , y_first = 0`
`array[1]`	`x_first = 20 , y_first = 0`
`array[2]`	`x_first = 30 , y_first = 40`

Initializing Arrays of Objects

> If the class has also a default constructor the programmer may define an array of objects as follows:

```
Point array[5]= {
        {10}, {20}, Point(30,40)
};
```

– Here, an array with ***5*** elements has been defined, but the list of initial values contains only ***3*** values, which are sent as arguments to the constructors of the first 3 elements.

– For ***the last two*** elements, the ***default constructor*** is called.

Initializing Arrays of Objects

> To call the default constructor for an object

```
Point array[5]={{10},Point(),{20}};
```

– Here, for the objects `array[1],array[3],array[4]` the default constructor is invoked.

> `Point array[5]={{10},,{20}}; // ERROR!`

Constructor Initializers

> Instead of assignment statements constructor initializers can be used to initialize data members of an object.

> Specially, to assign initial value to a constant member using the constructor initializer is the only way.

> Consider the class:

```
class C{
    const int CI;
    int x;
 public:
    C( ) {
        x = 0;
        CI = 0;
    }
};
```

```
class C{
    const int CI = 10 ;
    int x;
};
```

Solution

> The solution is to use a constructor initializer.

```
class C{
    const int CI;
    int x;
  public:
    C() : CI(0) {
          x = -2;
    }
};
```

> All data members of a class can be initialized by using constructor initializers.

```
class C{
    const int CI;
    int x;
  public:
    C( int a ) : CI(0), x (a)
      { }
};
```

Destructors

> The ***destructor*** is very similar to the constructor except that it is called automatically

1. when each of the objects goes out of scope

or

2. a dynamic object is deleted from memory by using the delete operator.

> A destructor is characterized as having the same name as the class but with a tilde '**~**' preceded to the class name.

> A destructor has no return type and receives no parameters.

> A class may have only one destructor.

Example

```
class String{
  int size; // Length of the string
  char *contents; // Contents of the string
 public:
  String(const char *); // Constructor
  void print(); // An ordinary function
  ~String(); // Destructor
};
```

> Actually, the standard library of C++ contains a **`string`** class.

> Programmers don't need to write their own string class. We write this class only to show some concepts.

```
String::String(const char *in_data) {
    cout<< "Constructor has been invoked" << endl;
    size = strlen(in_data);
    contents = new char[size +1];
    strcpy(contents, in_data);
}
void String::print() {
    cout << contents << " " << size << endl;
}
// Destructor
String::~String() {
    cout << "Destructor has been invoked" << endl;
    delete[] contents;
}
```

```
int main() {
    String string1("string 1");
    String string2("string 2");
    string1.print();
    string2.print();
    // destructor is called twice
    return 0;
}
```

Copy Constructor

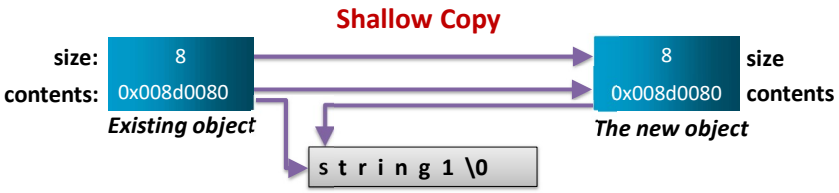
- It is a special type of constructors and used to copy the contents of an object to a new object during construction of that new object.
- The type of its input parameter is a reference to objects of the same type. It takes as argument a reference to the object that will be copied into the new object.
- The copy constructor is ***generated automatically*** by the compiler if the class author fails to define one.
- If the compiler generates it, it will simply copy the contents of the original into the new object as a byte by byte copy.

Copy Constructor

- For simple classes with no pointers, that is usually sufficient
- If there is a pointer as a class member so a ***byte by byte*** copy would copy the pointer from one to the other
  - Finally both points to the same allocated member!

Copy Constructor

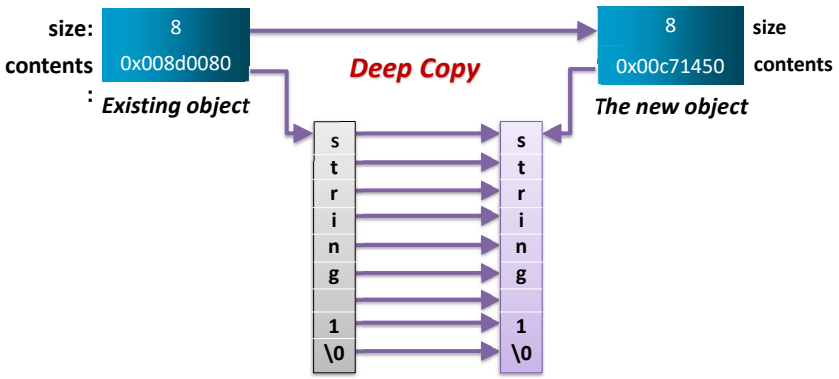
> For example the copy constructor, generated by the compiler for the String class will do the following job:



Copy Constructor

> The copy constructor, generated by the compiler can not copy the memory locations pointed by the member pointers.

> The programmer must write its own copy constructor to perform these operations.



Example: Copy constructor of String class

```
class String {
   int size;
   char *contents;
 public:
   String(const char *);
   String(const String &);
   void print();
   ~String();
};
String::String(const String &object_in) {
   cout<< "Copy Constructor has been invoked" << endl;
   size = object_in.size;
   contents = new char[size + 1];
   strcpy(contents, object_in.contents);
}
```

```
int main() {
   String my_string("string 1");
   my_string.print();
   String other = my_string;
   String more(my_string);
```

`const` Objects and `const` Member Functions

> The programmer may use the keyword const to specify that an object is not modifiable:

```
const TComplex cz(0,1); // constant object
```

> Any attempt to modify (to change the attributes) directly or indirectly (by calling a function) causes a compiler error.

`const` Objects and `const` Member Functions

- C++ compilers totally disallow any member function calls for **const** objects.
- The programmer may declare some functions as const, which do not modify any data of the object.
- Only const functions can operate on **const** objects.

```
// constant method
void print() const {
    cout << “complex number= “ << real << “, “ << img;
}
```

Example

```
void Point::print() const {
     cout << "X= " << x << ", Y= " << y << endl;
}
int main() {
    // constant point
    const Point cp(10,20);
    // non-constant point
    Point ncp(0,50);
    // OK. Const function operates on const object
    cp.print();
    // ERROR! Non-const function on const object
    cp.move(30,15);
    // OK. ncp is non-const
    ncp.move(100,45);
    return 0;
}
```

Explanation

> A const method can invoke only other const methods,

– A **const method** is **not allowed** to alter an object's state

- directly
- indirectly by invoking some non-const method

Another Example

```
class TComplex{
  float real,img;
 public:
  TComplex(float, float);    // constructor
  void print() const;          // const method
  void reset() {real=img=0;}   // non-const
method
};
```

```
TComplex::TComplex(float r=0,float i=0){
    real=r;
    img=i;
}
```

```
void TComplex::print() const {     // const method
     std::cout << "complex number= " << real << ", " << img;
}
```

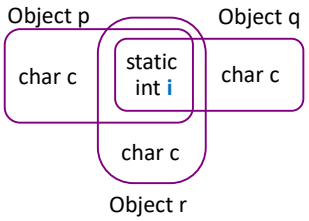
```
int main() {
 const TComplex cz(0,1);
 TComplex ncz(1.2,0.5)
 cz.print();  // OK
 cz.reset(); // Error !!!
 ncz.reset();  // OK
}
```

`static` Class Members

> Normally, each object of a class has its own copy of all data members of the class.

> In certain cases only one copy of a particular data member should be shared by all objects of a class. A static data member is used for this reason.

```
class A {
  char c;
  static int i;
};
int main() {
   A  p, q, r;
   :
}
```



`static` Class Members

> Static data members exist even no objects of that class exist.

> Static data members can be public or private.

> To access public static data when no objects exist use the class name and binary scope resolution operator.

for example int **A::i= 5;**

> To access private static data when no objects exist, a public **static member function** must be provided.

> They must be initialized **once** (and only once) at file scope.

Example

```
class A {
 char c;
 static int count; //  Number of created objects (static data)
 public:
       static int getCount(){return count;}
       A(){
            count ++;
            std::cout<< std::endl << "Constructor “ << count;
       }
      ~A(){
            count--;
            std::cout<< std::endl << "Destructor “ << count;
       }
};
int A::count=0;     // Allocating memory for number
```

Example

```
int main(){
 std::cout<<"\n Entering 1. BLOCK.....";
 A a,b,c;
 {
   std::cout<<"\n Entering 2. BLOCK.....";
   A d,e;
   std::cout<<"\n Exiting 2. BLOCK.....";
 }
 std::cout<<"\n Exiting 1. BLOCK.....";
}
```

Output

```
Entering 1. BLOCK.....
Constructor 1
Constructor 2
Constructor 3
Entering 2. BLOCK.....
Constructor 4
Constructor 5
Exiting 2. BLOCK.....
Destructor 5
Destructor 4
Exiting 1. BLOCK.....
Destructor 3
Destructor 2
Destructor 1
```

`const static` members

```
class Human {
  public:
      static const int HANDS = 2;
      static const int FINGERS = 10;
};
```

Passing Objects to Functions as Arguments

- Objects should be **returned by reference** unless there are compelling reasons to pass or return them by value.
- Returning by value can be especially inefficient in the case of objects.
- Recall that the object returned by value must be copied into stack and the data may be large, which thus wastes storage. The copying itself takes time.
- If the class contains a copy constructor the compiler uses this function to copy the object into stack.
- We should pass the argument by reference because we don't want an unnecessary copy of it to be created. Then, to prevent the function from accidentally modifying the original object, we make the parameter a const reference.

Passing Objects to Functions as Arguments

```
TComplex & TComplex::add(const TComplex& z){
    TComplex  result; // local object
    result.real = real + z.real;
    result.img = img + z.img;
    return result; // ERROR!
}
```

Remember, local variables can not be returned by reference.

Passing Objects to Functions as Arguments

```
TComplex TComplex::add(const TComplex& z){
    TComplex  result; // local object
    result.real = real + z.real;
    result.img = img + z.img;
    return result;
}
```

Passing Objects to Functions as Arguments

```
TComplex TComplex::add(const TComplex& z){
    TComplex  result; // local object
    result.real = real + z.real;
    result.img = img + z.img;
    return TComplex(result);
}
```

Avoiding Temporary Objects

- In the previous example, within the add function a temporary object is defined to add two complex numbers.
- Because of this object, constructor and destructor are called.
- Avoiding the creation of a temporary object within add() saves time and memory space.

```
TComplex TComplex::add(const TComplex& z)
    double  re_new,im_new;
    re_new = real + z.real;
    im_new = img + z.img;
    // Constructor is called
    return TComplex (re_new,im_new);
}
```

Avoiding Temporary Objects

- The only object that's created is the return value in stack, which is always necessary when returning by value.
- This could be a better approach, if creating and destroying individual member data items is faster than creating and destroying a complete object.

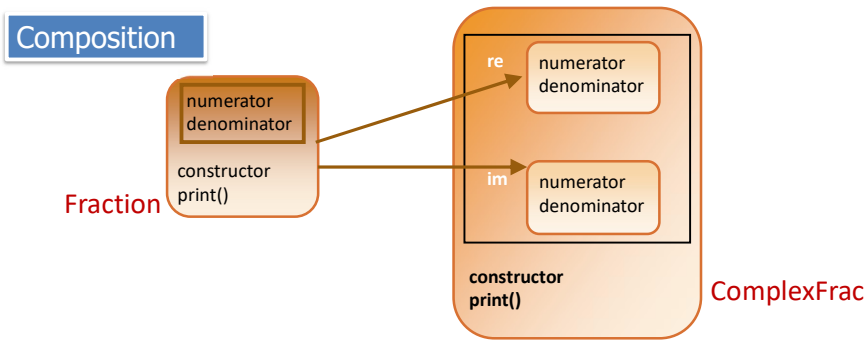
Avoiding Temporary Objects

```
void TComplex::add(const TComplex& z,TComplex &out)  {
   out.real = real + z.real;
   out.img = img + z.img;
}
```

Nesting Objects: Classes as Members of Other Classes

> A class may include objects of other classes as its data members.

> In the example, a class is designed (ComplexFrac) to define complex numbers. The data members of this class are fractions which are objects of another class (Fraction).



Composition & Aggregation

> The relation between Fraction and ComplexFrac is called "**has a relation**". Here, ComplexFrac has a Fraction (actually two Fractions).

> Here, the author of the class ComplexFrac has to supply the constructors of its object members (real , img) with necessary arguments.

> Member objects are constructed in the order in which they are declared and before their enclosing class objects are constructed.

> Example: A class to define fractions

```
class Fraction { // A class to define fractions
    int numerator, denominator;
 public:
    Fraction(int, int);     // CONSTRUCTOR
    void print() const;
};
Fraction::Fraction(int num, int denom)  {
    numerator = num;
    if (denom==0)
       denominator = 1;
    else
       denominator = denom;
    cout << "Constructor of Fraction" << endl;
}
void Fraction::print() const {
    cout << numerator << "/" << denominator << endl;
}
```

Example: A class to define complex numbers. It contains two objects as members

```
class ComplexFrac {
   Fraction real, img;
 public:
   ComplexFrac(int,int);
   void print() const;
 };
```

```
ComplexFrac::ComplexFrac(int re, int im) : real(re, 1) , img(im, 1) {
  ...
}
void ComplexFrac::print() const  {
   real.print();
   img.print();
}
int main()  {
    ComplexFrac cf(2,5);
    cf.print();
    return 0;
}
```

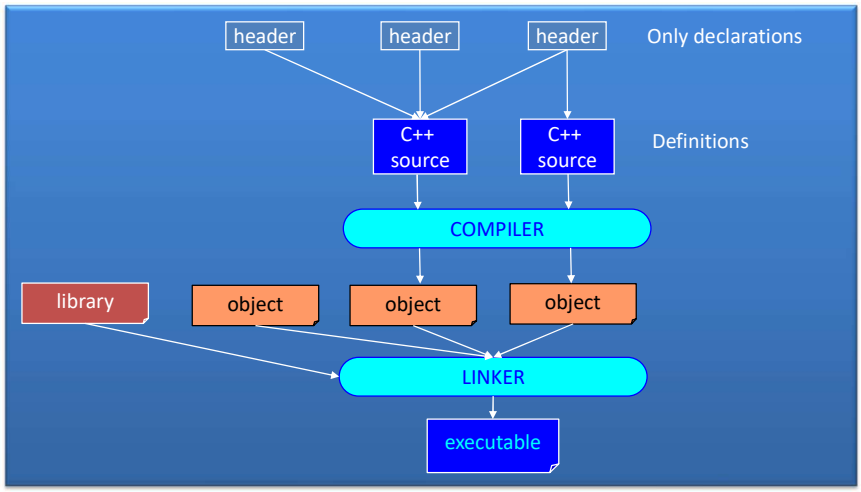
Data members are initialized

When an object goes out of scope, the destructors are called in reverse order: The enclosing object is destroyed first, then the member (inner) object.

Working with Multiple Files

- It is a good way to write each class or a collection of related classes in separate files.
- It provides managing the complexity of the software and reusability of classes in new projects.

Working with Multiple Files



Working with Multiple Files

- When using separate compilation you need some way to automatically compile each file and to tell the linker to build all the pieces along with the appropriate libraries and startup code into an executable file.
- The solution, developed on **Linux** but available everywhere in some form, is a program called make.
- Compiler vendors have also created their own project building tools.
- These tools ask you which files are in your project and determine all the relationships themselves.
- These tools use something similar to a **Makefile**, generally called a project file, but the programming environment maintains this file so you don't have to worry about it.

Working with Multiple Files

> The configuration and use of project files varies from one development environment to another, so you must find the appropriate documentation on how to use them (although project file tools provided by compiler vendors are usually so simple to use that you can learn them by playing around).

> We will write the example about fractions and complex numbers again.

> Now we will put the class for fractions and complex numbers in separate files.

CONSTRUCTORS AND DESTRUCTORS

QUIZ

Quiz #1

```
1 struct A
2 {
3    A(int x) : n(x) {}
4    int n;
5 };
6
7 int main(int argc, char** argv)
8 {
9    A a1 = 2;
10   a1 = 42;
10   A a2(2);
11   A a3(a2);
12   return 0;
13 }
```

Which lines should not compile?
a) none
b) 9
c) 10
d) 11
e) 9, 10, and 11

Quiz #2

```
struct Foo {
  Foo(int n) : x(n++), y(n++), z(n++) {}
  int x;
  int y;
  int z;
};
int main(int argc, char** argv) {
  Foo f(3);
  std::cout << "x: " << f.x << std::endl;
  std::cout << "y: " << f.y << std::endl;
  std::cout << "z: " << f.z << std::endl;
  return 0;
}
```

What gets printed for the value of z?
a) 3
b) 4
c) 5
d) The code does not compile
e) Undefined

Quiz #3

```
struct A
{
  A(A& a) { }
  A(double d) {}
  int val;
};
```

What is the maximum number of implicitly defined constructors that this struct will have?
a) 0
b) 1
c) 2
d) Undefined
e) Depends on the compiler

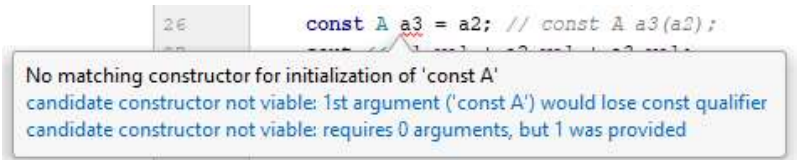
Quiz #4

```
struct A {
  A() : val(0) {}
  A(int v) : val(v) {}
  A(A& a) : val(a.val) {}
  int val;
};
int main(int argc, char** argv){
  const A a1;
  const A a2(5);
  const A a3 = a2; // const A a3(a2);
  cout << a1.val + a2.val + a3.val;
  return 0;
}
```

What gets printed?
a) 0
b) 5
c) 10
d) Undefined
e) The code does not compile

Answer #4

```
struct A {
    A() : val(0) {}
    A(int v) : val(v) {}
    A( A& a) : val(a.val) {}
    int val;
};
```



Answer #4

```
struct A {
    A() : val(0) {}
    A(int v) : val(v) {}
    A(const A& a) : val(a.val) {}
    int val;
};
```

Quiz #5

```
1 #include <iostream>
2
3 struct A
4 {
5   A(int& var) : r(var) {}
6   int &r;
7 };
8
9  int main(int argc, char** argv)
10 {
11   int x = 23;
12
13   A a1(x);
14
15   A a2 = a1;
16
17   a2 = a1;
18
19   return 0;
20 }
```

Which lines do not compile?

a) 13
b) 15
c) 17
d) 15 and 17
e) 13, 15, and 17

Answer #5

```
struct B {
    B(int &var) : r(var) {}

    int &r;
};
```

```
44        b2 = b1;
```

Object of type 'B' cannot be assigned because its copy assignment operator is implicitly deleted
copy assignment operator of 'B' is implicitly deleted because field 'r' is of reference type 'int &'